

Defining and composing big state machines

Victor Yodaiken

January 6, 2015

Abstract

A sequence function alternative representation of state machines.

1 Introduction

State machines are usually presented in terms of a set of events E , a set of states S , and a map $\delta : S \times A \rightarrow S$. Alternatively we can define a state variable as a function the set of finite sequences over E so that for any sequence s :

$$x = F(s)$$

The function F is an alternative presentation of a state machine in a way made precise in section 3. We can define a number of state variables (maybe a large number) all depending on the same sequence of events to describe a complex system. For example, given some sequence s that defines the current state, we might require that the number of processes that are executing code in a critical region never be more than one - where *Processes* and *ProgramCounter*(p) and *Critical* might all be state dependent.

$$|\{p \in \text{Process} : \text{ProgramCounter}(p) \in \text{Critical}\}| \leq 1$$

Section 2 shows techniques for defining and composing sequence functions. Section 3 shows the correspondence between sequence functions and standard state machine presentations. Examples of applications are in other papers. This work comes from a long process of attempting to refine initial intuitions about the utility of recursive sequence presentations of state machines and the use of general state machine products to model composition and parallel/concurrent computation[Yod09] and [Yod08]

2 Sequence functions

Many useful sequence functions can be defined by primitive recursion on sequences. Let *null* be the empty sequence and if s is a sequence and $a \in E$, let sa be the sequence obtained by appending a to s .

$$G(\text{null}) = x_0, \quad G(sa) = g(a, G(s)) \tag{1}$$

defines G on every finite sequence - assuming g is a previously defined function.

For example here is a mod k counter (or we could leave off the mod k and have an infinite state counter):

$$C_k(null) = 0,$$

$$C_k(sa) = \begin{cases} C_k(s) + 1 \bmod k & \text{if } a = \textit{increment}; \\ 0 & \text{if } a = \textit{reset}; \\ C_k(s) & \text{otherwise} \end{cases}$$

Standard function composition “hides state”:

$$D_k(s) = \begin{cases} 1 & \text{if } C_k(s) \neq 0; \\ 0 & \text{if otherwise.} \end{cases}$$

A simple composition produces a tuple of parallel sequence functions:

$$G(s) = (G_1(s), \dots G_n(s)) \quad (2)$$

For example to count mod 10 and mod 100 and mod 1000000 at the same time:

$$F(s) = (C_{10}(s), C_{100}(s), C_{1000000}(s))$$

so that $F(s) = (x, y, z)$ is a triple showing the three parallel counters.

To make components communicate requires a second level of recursion that is analogous to “simultaneous recursion” in classic primitive recursive function theory[Pet67]. In this case, recursion used to produce sequences for each component from the “global” sequence. For example, connect 2 mod k counters in a series so that counter 1 counts units, and counter 2 increments only when counter 1 rolls over to 0. Take the sequence s and define maps to s_1 and s_2 for the two components so that

$$H(s) = (C_k(s_1), C_k(s_2))$$

Now define the relationship between s and s_1 and s_2 be this:

- when we append a *reset* to s , append a *reset* to both s_1 and s_2 ,
- when we append *increment* to s , append *increment* to s_1 and
 - leave s_2 unchanged (if $C_k(s_1) < k - 1$) or
 - append *increment* to s_2 if $C_k(s_1) = k - 1$.

Then $H(s) = (n, m)$ indicates a count of $(n + m * k) \bmod k^2$.

More generally, suppose that we have a collection of n components each described by $G_i : E_i^* \rightarrow X_i, (0 < i \leq n)$ and a set of global events E . A map g has to be defined to specify the interaction between components so that $g(i, a, x_1 \dots x_n) = r_i$ gives the sequence of events component i sees when a single event a is appended to the global sequence and the component current state

values are given by $x_1 \dots x_n$. The function g determines a map $g^*(i, s) = s_i$ by recursion. Let $s \text{ concat } r$ be the sequence obtained by concatenating sequences s and r . We set $g^*(i, \text{null}) = \text{null}$ and then if $g(i, s) = s_i$ we define $g^*(i, sa) = s_i \text{ concat } r_i$ where $r_i = g(i, a, G_1(s_1) \dots G_n(s_n))$.

$$\begin{aligned} G(s) &= (G_1(g^*(1, s)) \dots G_n(g^*(n, s))) \\ \text{and } g^*(i, \text{null}) &= \text{null} \text{ and} \\ g^*(i, sa) &= g^*(i, s) \text{ concat } g(a, G_1(g^*(1, s)), \dots, G_n(g^*(n, s))) \end{aligned} \quad (3)$$

3 State machines

Obviously, finite state machines are an important class but I do not here assume state machines are finite.

A Moore type state machine is $M = (X, E, S, \sigma_0, \delta, \lambda)$ where $\sigma_0 \in S$ is the “start state” and $\delta : S \times E \rightarrow S$ is the transition function $\lambda : S \rightarrow X$ is the output map and $\lambda(\sigma)$ is the output of the state machine in state σ . A standard state machine can be considered to be a more machine where $\lambda(\sigma) = \sigma$.

Given Moore machine $M = (X, E, S, \sigma_0, \delta, \lambda)$ let E^* be the set of finite sequences over E including null and let

$$\delta^*(\sigma, \text{null}) = \sigma$$

and

$$\delta^*(\sigma, sa) = \delta(\delta^*(\sigma, s), a).$$

Let M^* be defined by

$$M^*(s) = \delta^*(\sigma_0, s)$$

Then $\lambda_M(M^*(s))$ is the output of M in the state reached by following s from the initial state.

Say M is an implementation of $G : E^* \rightarrow X$ if and only if $\lambda_M(M^*(s)) = G(s)$ for all $s \in E^*$.

Say G is finite state if and only if there is a M that implements G where the state set of M is finite.

If E and X are finite sets and $g : E \times X \rightarrow X$ then G defined by $G(\text{null}) = x_0 \in X$ and $G(sa) = g(a, G(s))$ is finite state.

Suppose that M_1, \dots, M_n implement G_1, \dots, G_n and $M_i = (X_i, E_i, S_i, \sigma_{0,i}, \delta_i, \lambda_i)$. Define G using definition form 3. Then we can construct a product of the M_i that implements G as follows:

$$\text{The state set } S = S_1 \times \dots \times S_n$$

$$\text{The output map } \lambda((\sigma_1 \dots \sigma_n) = (\lambda_1(\sigma_1), \dots, \lambda_n(\sigma_n))$$

$$\text{The transition map } \delta(\sigma, a) = (\dots \delta_i^*(\sigma_i, g(i, a, \lambda(s))) \dots) \text{ where } \sigma = (\sigma_1, \dots, \sigma_n)$$

Clearly, G is finite state if all the G_i are finite state. The state machine product here is well known. See for example [Gec86].

There is a type of state machine product often called a “cascade” product where the flow of information does not include any cycles. Note that if for all $s \neq null$ we have $\delta^*(\sigma, s) \neq \sigma$ then the state machine has only trivial cycles. This has some structural implications [Pin86]. But cascades describe state machine products with only trivial cycles - something different, although there may well be a relationship. The Krohn-Rhodes theorem [Arb68] relates cascade products of state machines to simple groups and the well known Jordan-Hlder theorem.

In a definition of type 3 consider whether $g(i, x_1 \dots, x_n)$ depends on the j^{th} element x_j or not. For example, if we define $g(i, a, x) = \langle a \rangle$ then g and i do not depend on any of the x_j . If there is a partial order R on $\{1 \dots, n\}$ so that for each g and i do not depend on any j with iRj , then say that the composite system is a “cascade”.

References

- [Arb68] Michael A. Arbib. *Algebraic theory of machines, languages, and semi-groups*. Academic Press, 1968.
- [Gec86] Ferenc Gecseg. *Products of Automata*. Monographs in Theoretical Computer Science. Springer Verlag, 1986.
- [Pet67] Rozsa Peter. *Recursive functions*. Academic Press, 1967.
- [Pin86] J.E. Pin. *Varieties of Formal Languages*. Plenum Press, New York, 1986.
- [Yod08] Victor Yodaiken. State and history in operating systems. *CoRR*, abs/0805.2749, 2008.
- [Yod09] Victor Yodaiken. Primitive recursion and state machines. *CoRR*, abs/0907.4169, 2009.